Top of Form



**Introduction to C# Programming: Lesson 4**

**Chapter 1**

**Introduction**

Now that you know how to use variables and how to get input from the user, you can write very useful programs. The next two lessons will give you a chance to practice what you have learned using methods. I usually use methods as a first step to understanding object-oriented programming. Methods allow for code reuse, which is a wonderful thing.

In Lesson 4, I plan to start you out with methods by working with some that have already been written for you. Once you are comfortable with calling these standard methods, you will then learn how to write and call your own. In the process of learning all of this, you will learn about passing values to and from these methods.  
  
  
  
  
**Chapter 2**

**Built-in Methods from Class Math**

The first thing we must do is define the term *method*. A *method* is simply a group of one or more lines of code that perform some task and are given their own name. Other programming languages call these things functions, procedures, or subroutines. Methods are easy to recognize in a program by the parentheses that immediately follow the method's name. Already you have done some work with the Write and WriteLine methods. You know that when you put a literal string or a variable name inside the parentheses, that value is displayed on the screen. The thing that you put inside the parentheses is known as an *actual argument*. Each argument represents information that will be used by the method to perform its task. It is important to note that not all methods require actual arguments. You have already seen such a method, Main.

Did you ever wonder about how the WriteLine method worked? Well, when the C# language was being developed, someone probably thought that programmers would want to display values to the screen very often. Therefore, instead of making the programmer write this code every time they wanted to print something, they wrote the code inside a method that we can all use any time we want.

As you have seen, you can make a method perform its task by calling, or invoking, the method in a program. You do this by writing the name of the method, and any arguments necessary, in a statement. This idea of calling on a method in C# may seem a little foreign to you, but you should be familiar with the concept. Have you ever ordered a pizza on the telephone? You called the pizza place and provided them with your name, address, and what kind of toppings you wanted on your pizza. Similar to what you do with a method, you called the pizza place to perform a task for you. Notice how, before the restaurant could make the pizza for you, it needed some important information. This information is similar to the actual arguments that you will be providing to your method.

When the C# compiler encounters a line of code that contains a call to a method, like WriteLine, it finds the place where the method is defined. Next, the actual arguments are passed to the method, and then the code inside the method is executed.

What the program does after its task is completed is dependent on whether the method is a value-returning or a void method. A *value-returning method* is a method that passes back, or returns, a value back to the statement that called it. A *void method* is a method that performs some task, but it does not return a value to that calling statement. The remainder of this lesson will focus on value-returning methods, while the next lesson will focus on void methods.

An example of a value-returning method that you have already seen is the ReadLine method. Recall that this method, which does not take any actual arguments, allows the user to enter information from the keyboard. Then the method passes back, or returns, the user's response as a string.

As you might expect, there are many other methods available for you to use. To give you a little bit of practice with some built-in methods, I will turn to a group of methods that provide mathematical functions. When the mathematical operators were presented earlier in the course, you may have been a little curious about some of the more advanced mathematical operations.

Specifically, I usually get lots of questions about how to do exponents. I have students who want to do things like compute the hypotenuse of a triangle. For those geometry lovers out there, you will recall that the hypotenuse is the square root of the first side squared plus the second side squared.

To do such a calculation, you would need to be able to do exponents and square roots. These methods are provided in a class appropriately called Math. Class Math contains a number of methods and a couple of constant variables. All of these things are declared as static. We will discuss what *static* means later. For now, you just need to know that in order to use static variables or methods, you must precede the call by the class name and the dot operator (.). For example, to calculate the square root of 4.5, you would call the square root method, named Sqrt, by typing Math.Sqrt(4.5);

Both of the square root (Sqrt) and exponent (Pow) methods are value-returning, meaning that a value is returned to the calling statement. Remember, when a value-returning method is called, the calling statement must do one of three things with the result: store the value, print the value, or use the value in an equation. Let's look at an example of the square root function:

double answer = Math.Sqrt(9.5);

This statement will pass the value 9.5 to the Sqrt method, which will then calculate the square root and return the result. Notice that the Sqrt method requires a single actual argument. This argument is the value that you want to calculate the square root of.

The actual argument need not be a literal value. Instead, a variable can be used. Here is another example, which stores the sum of 10.2 and square root of the variable *number* in a variable called *result*, which has already been declared.

result = 10.2 + Math.Sqrt(number);

The method used for calculating exponents is called Pow. Pow is a method that takes two actual arguments. The first is the base value, and the second is the value that you want the base value raised to. For example, to print out the result of two raised to the third power, you would type:

System.Console.Out.WriteLine(Math.Pow(2, 3));

Notice something interesting in this statement. There is a method call inside a method call. This is perfectly legal in C#. As you would expect, just like in math class, you start executing the code in the innermost parentheses first. Therefore, the compiler will pass the values 2 and 3 to the Pow method. This method calculates the result as 8 and passes that value back to the statement. Next, the value 8 is passed to the WriteLine method. Finally, this method displays the value on the screen. Lots of work for one little line of code! Feel free to use method calls inside of method calls as often as you wish. However, I usually recommend not doing too much until you are more comfortable with the C# language.

C# also provides methods for calculating things like absolute value (Abs), sine (Sin), cosine (Cos), and tangent (Tan). There are also methods provided for calculating the maximum of two numbers (Max), the minimum of two numbers (Min), and rounding a number (Round). Notice how the method names are very similar to what the method actually does. This will certainly make remembering the method names easy when you are writing your code. Remember this principle later when it comes time for you to name your methods.  
  
  
  
  
**Chapter 3**

**Writing Your First Method**

So far, you have learned about and used some methods that have been written for you. Now you are ready to move on and write your own value-returning method. The first thing to write when you write your method is the method header. The method header is very important to C# because it specifies a number of important things. Below is the general format for a method header:

*access modifier*> *optional static modifier*> *return type*> *method name*> ( *parameter list*>)

The first thing that you will type in the method header is the *access modifier*. We will discuss this in more detail when we discuss object-oriented programming in more detail. For now, just create all of your methods using the modifier *public*. The next word you see is the keyword *static*. Again, we will discuss this more when we get deeper into object-oriented programming. For now, be sure to write the keyword *static* in your method headers. The third word in the method header is the data type of the value that you are returning. Although the return value can be of any C# type, for this course you will probably return something like int, double, or string. The next word in the header is the name of the method. Be sure to follow the naming conventions mentioned earlier. The final part of the method header is the parameter list.

Recall from earlier in the lesson that the calling statement is able to pass values to the method. This is done by placing the actual arguments inside the parentheses in the method call. The values of the arguments are then copied and stored in the variables declared in the parameter list. The following line of code is a method header for a method called Show that will take one int as parameters and return an int.

public static int Show (int value)

Notice how the variable is declared in the parameter list. First, the data type is listed, followed by the name of the variable. If two or more values are to be passed, then each variable will be declared the same way, with each declaration being separated by a comma. For example:

public static int Sum (int number1, int number2)

In order for a statement to call this method, the statement would need to list the name of the method, Sum, followed by two ints as actual parameters. The value of the first int in the call statement would be passed into the first variable listed, number1. Likewise, the value of the second int in the call statement would be passed into the second variable listed, number2. Every call statement's arguments must exactly match the parameters in the parameter list in both number and data type. Therefore, if you tried to call the above method with the statement by passing it three ints, the program would not work. Similarly, you could not try to pass this method two doubles.

Not every method requires information to be passed to it. Therefore, not every method header will have variable declarations in the parameter list. If this were the case, then there would be no text inside the parentheses. Note, however, that the parentheses are still required. An example of such as method header is shown below:

public static double NewMethod( )

I find that often times writing the method header is the hardest part of writing a method. To help solve this problem, I usually require my students to write three lines of comments before every method that they write. The first comment line should start with the word *task*, followed by the task that this method will accomplish. The second line should start with the word *incoming*. Next, list all of the items that this method will need to accomplish its task. This will help you figure out what needs to go in the parameter list. The final comment line should start with the word *outgoing*. It will then describe what, if anything, is going to be returned by the method. Notice how the comments directly correlate with the different parts of the method header.

Following the method header is the method body. As you may recall from earlier lessons, a method body begins with an opening curly brace ({) and ends with a closing curly brace (}). Inside the body of the method is where you type the statements that are to be performed when the method is called.

One very important part of your value-returning method is the return statement. The *return statement* is the way that your method passes a value back to the calling statement. The return statement is the signal to the compiler that the method is finished. The value will be immediately returned, and execution continues with the calling statement. That means that any statements after the return statement will not be executed.

All of these words may be a little confusing to you at this point. Hopefully, the following program will help to clear up any such confusion. Below is a method named Sum that receives two integers, adds them, and returns the result.

public static int Sum(int number1, int number2)

{

int sum;

sum = number1 + number2;

return sum;

}

The next section will discuss a little more about how the calling statement and method communicate with one another. This section will also explain the concept of variable scope and show a complete program as an example.  
  
  
  
  
**Chapter 4**

**Method Calls and Variable Scope**

OK, so you got your first method written, and you are ready to use it. Although writing the method may have been a little foreign to you, the good news is that calling your own method is about the same as calling any other method.

Remember that a method call begins with the name of the method, followed by the actual arguments enclosed in parentheses. That means that to call the Sum function defined earlier, pass the values 2 and 3, and store the result in a variable named *result*, you would type:

result = Sum(2, 3);

Don't forget that since Sum is a value-returning method, the value that is returned must be printed, stored, or used in an equation. Now, you might recall that when we defined Sum, we made it a static method. You may also remember that earlier in the lesson I said that when you call on static methods, you must include the name of the class where the method is defined. We did this when we called the Sqrt method from class Math. Well, Sum was a static method, but the class name was not used in the above statement. This can be done only if the method is defined in the same class as the calling statement. A common scenario would be having a statement inside method Main that calls on another static method, possibly Sum, which is defined in the same class.

A class can have any number of methods; just remember the rules about where things are defined. The class body starts with an opening curly brace ({) and ends with a closing curly brace (}). A method can be defined anywhere inside the body of the class. Just be sure to end your first method before you start defining your second method, because you cannot define one method inside of another method.

To help clear up the idea of having one class with multiple methods, an example is provided below. You may find it useful to sit down, type this program into Notepad, and compile and run it. As you type, read and think about each line of code, one at a time. After all, this is how the compiler reads the code—one line at a time. It is just that the compiler reads the line really, really fast!

using System;

public class Lesson4

{

public static void Main()

{

string strFirst, strSecond;

int first, second, result;

Console.Out.Write("Enter a number: ");

strFirst = Console.ReadLine();

first = Convert.ToInt32(strFirst);

Console.Out.Write("Enter a second number: ");

strSecond = Console.ReadLine();

second = Convert.ToInt32(strSecond);

result = Sum(first, second);

Console.Out.WriteLine("The sum is: " + result);

}

public static int Sum(int number1, int number2)

{

int answer = number1 + number2;

return answer;

}

}

When you run this program, it will prompt the user for two numbers. It then passes those values to the Sum method, which calculates and returns the sum of the two numbers. Finally, the Main method displays this result on the screen.

There are a few things about this program that I want to point out. Starting at the top, you may have noticed the statement, using System;. You will find as you write larger and larger programs, you end up typing the word *System* an awful lot. To help make our lives easier, C# has provided a shortcut for us. Since the Console class resides in what is called the System namespace, we can tell the compiler to always go to that namespace when it encounters the Console class by typing the using statement before the class header.

Another obvious part of this program is the fact that there is another method in addition to Main. You should be aware that methods can be listed in any order you wish. That is, you could have typed in the Sum method first and then the Main method; the compiler does not care. Since program execution always starts in Main, I generally start my programs with the definition for Main. This also makes it easier for me to find it.

The final point I want to make about this program is variable names. Did you notice that the call statement in the Main method uses variables named *first* and *second*. On the other hand, the Sum method uses variables named *number1* and *number2*. This is perfectly acceptable in C#. That is, the argument variable names do not have to match the parameter variable names. In fact, I prefer that my students do *not* use the same names for variables in different methods. I often find that students who do use the same names end up confusing themselves, and this is not good. After all, they are different variables.

This brings me to the concepts of variable scope and lifetime. A variable's scope indicates which parts of a program can access that variable. A variable can have either local or global scope. The variable's scope is determined by where the variable is declared. For now, we will only create variables with local scope. That means that any variable declared inside the Main method can only be used in the Main method. From the program example above, this means that the Main method does not have access to the variables number1, number2, or answer. When I say, "have access to," I mean that the statements inside Main are not allowed to print the value of number1 or use its value in an equation.

A variable's lifetime refers to how long that spot of memory is reserved for the variable. Again, the variable's lifetime is determined by the location of the variable declaration. *Local variables*, those declared inside of a method, remain in memory until the method ends. So, let's look at the program again in greater detail.

The program begins by executing the first statement in the Main method. This statement creates two local string variables. The next statement creates three local int variables. Next, there are three lines of code that prompt the user for a number, store the response as a string, and then convert the string to an int for storage in the first variable. The next three lines of code do the same thing to get a value for the second variable. The next line of code calls on the Sum method and passes the values of the first and second variables to that method.

Once the Sum method is called, execution in the Main method stops until the Sum method is finished executing its code. In the Sum method header, variables number1 and number2 are created. These are local variables that will only be active inside the Sum method. The value of *first* is copied into the number1 variable, and the value of second is then copied into the number2 variable. Execution in the Sum method continues with the creation of the local variable answer, which stores the sum of number1 and number2. The final statement in the Sum method is the statement that tells the compiler to return the value of the answer variable to the calling statement.

When the value of *answer* is returned, it is then copied into the result variable. Since the Sum method is finished executing, the values of the variables number1, number2, and answer are deleted, and the memory is freed up for later use. Finally, the Main method prints to the screen a statement telling the user the sum of the two numbers.

As you can see, there is a lot of work done by this simple program. In addition to the calculations, which are easy for us, there is also the need to create variables, store values, display results, and destroy variables. This is all very complicated, but thanks to the C# language and the C# compiler, writing such a program is not very difficult. At first, you may have a difficult time following the flow of programs with many method calls, but you will get used to it. You will probably even find that writing your code in methods actually makes programming much easier. It helps you to better organize and reuse your code. You will also find that it makes debugging your programs much easier.  
  
  
  
  
**Chapter 5**

**Summary**

At this point in the course, you have taken a big step toward being a better object-oriented programmer. It is easy to write programs that only have a Main method. In fact, most programs can be written this way. However, as your programs become more and more complex, you will find that it is difficult to follow the actions of your code. You will find that there is more need to organize the code. You will also find that you will want to run a section of code again later in a program. For example, imagine if you wanted the program from the lesson to be run two times. That is, you want to ask the user for two numbers, display their sum, and then ask for two different numbers and display the new sum. Without methods, you would need to type the code from Sum into the Main method twice. By using methods, you can now just add a second call to the Sum method. This might not seem that exciting to you, probably because the Sum method really just consists of one line of code. However, imagine if you had a method that needed 1,000 lines of code in order to calculate a result. You certainly would think that replacing 1,000 lines with one line would be a great thing!

In the next lesson, you will learn how to write and call a method that does not return a value. You will also learn how to pass the address of a variable instead of its value. Once again, this will help to make your programs more useful.  
  
  
  
**Supplementary Material**

|  |
| --- |
| [All Methods and Variables in Class Math](http://msdn.microsoft.com/en-us/library/system.math.aspx)  http://msdn.microsoft.com/en-us/library/system.math.aspx |
| This site lists all of the variables and method methods that are in class Math. Each method header is given. |

**FAQs**   
  
**Q:** The lesson states that I can use the same variable name in multiple methods, but it is not advised. Sometimes I cannot think of a different name. Is it ok then?  
  
**A:** I know exactly what you mean. Often times I will have a variable called total in the Main method. You are correct when you say that you can use the same name, but it is not advised. So what I do is, use the underscore before the variable name. That way Main can have a variable called total and another method can have a varible named \_total. I also suggest that people just number their variables. That way you can have total, total2, total3, etc.

**Assignment**   
  
  
Write a program that will calculate the hypotenuse of a right triangle. Have your Main method get the lengths of the two sides. It should then call on a method called *calcHypotenuse*. This method should take two doubles as parameters and return the hypotenuse as a double. Remember that to calculate the hypotenuse, you add the square of the first side to the square of the second side. Take the square root of that sum to get the hypotenuse. For example, to calculate a right triangle with sides of length 3 and 4:

hypotenuse = square root ( 3^2 + 4^2 ) = 5

[Click here for solution: **RightTriangle.zip**](https://api.ed2go.com/CourseBuilder/2.0/images/resources/prod/cpb-0/RightTriangle.zip)

|  |
| --- |
| Copyright © 1997 - 2012 Education To Go. All rights reserved. The material on this site cannot be reproduced or redistributed unless you have obtained prior written permission from Education To Go. Education To Go and ed2go are registered trademarks of Education To Go, a part of Cengage Learning. |

Bottom of Form